Genetic Algorithms for Word Problems in Partially Commutative Groups

Matthew J. Craven

Mathematical Sciences, University of Exeter,
North Park Road, Exeter EX4 4QF, UK.

Abstract. We describe an implementation of a genetic algorithm on partially commutative groups and apply it to the double coset search problem on a subclass of groups. This transforms a combinatorial group theory problem to a problem of combinatorial optimisation. We obtain a method applicable to a wide range of problems and give results which indicate good behaviour of the genetic algorithm, hinting at the presence of a new deterministic solution and a framework for further results.

1 Introduction

1.1 History and Background

Genetic algorithms (hereafter referred to as GAs) were introduced by Holland [4] and have enjoyed a recent renaissance in many applications including engineering, scheduling and attacking problems such as the travelling salesman and graph colouring problems. However, the use of GAs in group theory [1,7,8] has been in operation for a comparatively short time.

This paper discusses an adaptation of GAs for word problems in combinatorial group theory. We work inside the Vershik groups [11], a subclass of partially commutative groups (also known as graph groups [10] and trace groups). We omit a survey of the theory of the groups here and focus on certain applications.

There exists an explicit solution for many problems in this setting. The biautomaticity of the partially commutative groups is established in [10], so as a corollary the conjugacy problem is solvable. Wrathall [12] gave a fast algorithm for the word problem based upon restricting the problem to a monoid generated by group generators and their formal inverses. In [13], an algorithm is given for the conjugacy problem; it is linear time by a stack-based computation model.

Our work is an experimental investigation of GAs in this setting to determine why they seem to work in certain areas of combinatorial group theory and to determine bounds for what happens for given problems. This is done by translating given word problems to ones of combinatorial optimisation.

1.2 Partially Commutative Groups and Vershik Groups

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set and define the operation of multiplication of $x_i, x_j \in X$ to be the juxtaposition $x_i x_j$. As in [13], we specify a *partially*

commutative group $G(X)$ by X and the collection of all elements from X that *commute*; that is, the set of all pairs (x_i, x_j) such that $x_i, x_j \in X$ and $x_i x_j = x_j x_i$. For example, take $X = \{x_1, x_2, x_3, x_4\}$ and suppose that $x_1 x_4 = x_4 x_1$ and $x_2 x_3 = x_3 x_2$. Then we denote this group $G(X) = \langle X : [x_1, x_4], [x_2, x_3] \rangle$.

The elements of X are called *generators* for $G(X)$. Note that for general $G(X)$ some generators commute and some do not, and there are no other non-trivial relations between the generators. We concentrate on Vershik groups, a particular subclass of the above groups. For a set X with n elements as above, the *Vershik group of rank* n over X is given by

$$V_n = \langle X : [x_i, x_j] \text{ if } |i - j| \geq 2 \rangle .$$

For example, in the group V_4 the pairs of elements that commute with each other are $(x_1, x_3), (x_1, x_4)$ and (x_2, x_4). We may also write this as $V(X)$ assuming an arbitrary set X. The elements of V_n are represented by group *words* written as products of generators. The *length*, $l(u)$, of a word $u \in V_n$ is the minimal number of single generators from which u can be written. For example $u = x_1 x_2 x_4 \in V_4$ is a word of length three. We use x_i^μ to denote μ successive multiplications of the generator x_i; for example, $x_2^4 = x_2 x_2 x_2 x_2$. Denote the empty word $\varepsilon \in V_n$.

For a subset, Y, of the set X we say the Vershik group $V(Y)$ is a *parabolic subgroup* of $V(X)$. It is easily observed that any partially commutative group G may be realised as a subgroup of a Vershik group V_n of sufficiently large rank n.

Vershik [11] solved the word problem in V_n by means of reducing words to their *normal form*. The Knuth-Bendix normal form of a word $u \in V_n$ of length $l(u)$ may be thought of as the "shortest form" of u and is given by the unique expression

$$\overline{u} = x_{i_1}^{\mu_1} x_{i_2}^{\mu_2} \ldots x_{i_k}^{\mu_k}$$

such that all $\mu_i \neq 0, l(\overline{u}) = \sum |\mu_i|$ and

i) if $i_j = 1$ then $i_{j+1} > 1$;
ii) if $i_j = m < n$ then $i_{j+1} = m - 1$ or $i_{j+1} > m$;
iii) if $i_j = n$ then $i_{j+1} = n - 1$.

The name of the above form follows from the Knuth-Bendix algorithm with ordering $x_1 < x_1^{-1} < x_2 < x_2^{-1} < \ldots < x_n < x_n^{-1}$. We omit further discussion of this here; the interested reader is referred to [6] for a description of the algorithm.

The algorithm to produce the above normal form is essentially a restriction of the stack-based (or heap-based) algorithm of [12] to the Vershik group, and we thus conjecture that the normal form of a word $u \in V_n$ may be computed efficiently in time $O(l(u) \log l(u))$ for the "average case". From now on we write $\overline{u}$ to mean the normal form of the word u. For a word $u \in V_n$, we say that

$$RF(u) = \{x_i^\alpha : l(\overline{u x_i^{-\alpha}}) = l(\overline{u}) - 1, \alpha = \pm 1\}$$

is the *roof of* u and

$$FL(u) = \{x_i^\alpha : l(\overline{x_i^{-\alpha} u}) = l(\overline{u}) - 1, \alpha = \pm 1\}$$

is the *floor of* u. The roof (and floor) of u correspond to the generators which may be cancelled after their inverses are juxtaposed to the right (and left) end of u to create the word u' and u' is reduced to its normal form $\overline{u'}$. For example, if $u = x_1^{-1}x_2x_6x_5^{-1}x_4x_1$ then $RF(u) = \{x_1, x_4\}$ and $FL(u) = \{x_1^{-1}, x_6\}$.

2 Statement of Problem

Given a Vershik group V_n and two words a, b in the group, we wish to determine whether a and b lie in the same double coset with respect to given subgroups. In other words, consider the following problem:

The Double Coset Search Problem (DCSP) Given two parabolic subgroups $V(Y)$ and $V(Z)$ of a Vershik group V_n and two words $a, b \in V_n$ such that $b \in V(Y)\, a\, V(Z)$, find words $x \in V(Y)$ and $y \in V(Z)$ such that $b = xay$.

We attack this group-theoretic problem by transforming it into one of combinatorial optimisation. In the following exposition, an *instance* of the DCSP is specified by a pair (a, b) of given words, each in V_n, and the notation $\mathcal{M}((a, b))$ denotes the set of all *feasible solutions* to the given instance. We will use a GA to iteratively produce "approximations" to solutions to the DCSP, and denote an "approximation" for a solution $(x, y) \in \mathcal{M}((a, b))$ by $(\chi, \zeta) \in V(Y) \times V(Z)$.

Combinatorial Optimisation DCSP

Input: Two words $a, b \in V_n$.
Constraints: $\mathcal{M}((a, b)) = \{(\chi, \zeta) \in V(Y) \times V(Z) : \chi a \zeta \doteq b\}$.
Costs: The function $C((\chi, \zeta)) = l(\chi a \zeta b^{-1}) \geq 0$.
Goal: Minimise C.

The cost of the pair (χ, ζ) is a non-negative integer imposed by the above function C. The length function defined on V_n takes non-negative values; hence an *optimal solution* for the instance is a pair (χ, ζ) such that $C((\chi, \zeta)) = 0$. Therefore our goal is to minimise the cost function C.

As an application of our work, note that the Vershik groups are inherently related to the braid groups, a rich source of primitives for algebraic cryptography. In particular, the DCSP in the Vershik groups is an analogue of an established braid group primitive. The reader is invited to consult [5] for further details.

In the next section we expand these notions and detail the method we use to solve this optimisation problem.

3 Genetic Algorithms on Vershik Groups

3.1 An Introduction to the Approach

For brevity we do not discuss the elementary concepts of GAs here, but refer the reader to [4,9] for a discussion of GAs and remark that we use standard terms such as *cost-proportionate selection* and *reproductive method* in a similar way.

We give a brief introduction to our approach. We begin with an initial population of "randomly generated" pairs of words, each pair of which is treated as an approximation to a solution $(x, y) \in \mathcal{M}((a, b))$ of an instance (a, b) of the DCSP. We explicitly note that the GA does not know either of the words x or y. Each pair of words in the population is ranked according to some cost function which measures how "closely" the given pair of words approximates (x, y). After that we systematically imitate natural selection and breeding methods to produce a new population, consisting of modified pairs of words from our initial population. Each pair of words in this new population is then ranked as before. We continue to iterate populations in this way to gather steadily closer approximations to a solution (x, y) until we arrive at a solution (or otherwise).

3.2 The Representation and Computation of Words

We work in V_n and two given parabolic subgroups $V(Y)$ and $V(Z)$, and wish the GA to find an exact solution to a posed problem. We naturally represent a group word $u = x_{i_1}^{\mu_1} x_{i_2}^{\mu_2} \dots x_{i_k}^{\mu_k}$ of arbitrary length by a string of integers, where we consecutively map each generator of the word u as follows:

$$x_i^{\epsilon_i} \to \begin{cases} +i & \text{if } \epsilon_i = +1 \\ -i & \text{if } \epsilon_i = -1 \end{cases}$$

For example, if $u = x_1^{-1} x_4 x_2 x_3^2 x_7 \in V_7$ then u is represented by the string `-1 4 2 3 3 7`. In this context the length of u is equal to the number of integers in its string representation. We define a *chromosome* to be the GA representation of a pair (χ, ζ) of words, and note that each word is naturally of variable length. Moreover a *population* is a multiset of a fixed number p of chromosomes. The GA has two populations in memory, the *current population* and the *next generation*. As with traditional GAs the current population contains the chromosomes under consideration at the current iteration of the GA, and the next generation has chromosomes deposited into it by the GA which form the current population on the next iteration. A *subpopulation* is a submultiset of a given population.

We use the natural representation for ease of algebraic operation, acknowledging that faster or more sophisticated data structures exist, for example the stack-based data structure of [13]. However we believe the simplicity of our representation yields relatively uncomplicated reproductive algorithms. In contrast, we believe a stack-based data structure yields reproductive methods of considerable complexity. We give our reproductive methods in the next subsection.

Besides normal form reduction of a word u we use *pseudo-reduction* of u. Let $\{ x_{i_{j_1}}, x_{i_{j_1}}^{-1}, \dots, x_{i_{j_m}}, x_{i_{j_m}}^{-1} \}$ be the generators which would be removed from u if we were to reduce u to normal form. Pseudo-reduction of u is defined as simply removing the above generators from u. There is no reordering of the resulting word (as with normal form). For example, if $u = x_6 x_8 x_1^{-1} x_2 x_8^{-1} x_2^{-1} x_6 x_4 x_5$ then its *pseudo-normal form* is $\tilde{u} = x_6 x_1^{-1} x_6 x_4 x_5$ and the normal form of u is $\overline{u} = x_1^{-1} x_4 x_6^2 x_5$. Clearly, we have $l(\tilde{u}) = l(\overline{u})$. This form is efficiently computable, with complexity at most that of the algorithm used to compute the normal form $\overline{u}$. Note, a word is not assumed to be in any given form unless otherwise stated.

3.3 Reproduction

The following reproduction methods are adaptations of standard GA reproduction methods. The methods act on a subpopulation to give a child chromosome, which we insert into the next population (more details are given in section 5).

1. Sexual (*crossover*): by some selection function, input two parent chromosomes c_1 and c_2 from the current population. Choose one random segment from c_1, one from c_2 and output the concatenation of the segments.
2. Asexual: input a parent chromosome c, given by a selection function, from the current population. Output one child chromosome by one of the following:
   (a) *Insertion* of a random generator into a random position of c.
   (b) *Deletion* of a generator at a random position of c.
   (c) *Substitution* of a generator located at a random position in c with a random generator.
3. Continuance: return several chromosomes $c_1, c_2, \ldots, c_m$ chosen by some selection algorithm, such that the first one returned is the "fittest" chromosome (see the next subsection). This method is known as *partially elitist*.
4. Non-Local Admission: return a random chromosome by some algorithm.

With the exception of continuance, the methods are repeated for each child chromosome required.

3.4 The Cost Function

In a sense, a cost function induces a partial metric over the search space to give a measure of the "distance" of a chromosome from a solution. Denote the solution of an instance of the DCSP in section 2 by (x, y) and a chromosome by (χ, ζ). Let $E(\chi, \zeta) = \chi a \zeta b^{-1}$; for simplicity we denote this expression by E. The normal form of the above expression is denoted $\overline{E}$. When (χ, ζ) is a solution to an instance, we have $\overline{E} = \varepsilon$ (the empty word) with defined length $l(\overline{E}) = 0$.

The cost function we use is as follows: given a chromosome (χ, ζ) its cost is given by the formula $C((\chi, \zeta)) = l(\overline{E})$. This value is computed for every chromosome in the current population at each iteration of the GA. This means we seek to minimise the value of $C((\chi, \zeta))$ as we iterate the GA.

3.5 Selection Algorithms

We realise continuance by roulette wheel selection. This is cost proportionate. As we will see in Algorithm 2, we implicitly require the population to be ordered best cost first. To this end, write the population as a list $\{(\chi_1, \zeta_1), \ldots, (\chi_p, \zeta_p)\}$ where $C(\chi_1, \zeta_1) \leq C(\chi_2, \zeta_2) \leq \ldots \leq C(\chi_p, \zeta_p)$. Then the algorithm is as follows:

Algorithm 1 (Roulette Wheel Selection)

INPUT: *The population size p; the population chromosomes (χ_i, ζ_i); their costs $C((\chi_i, \zeta_i))$; and n_s, the number of chromosomes to select*

OUTPUT: *n_s chromosomes from the population*

1. *Let* $W \leftarrow \sum_{i=1}^{p} C((\chi_i, \zeta_i))$;
2. *Compute the sequence* $\{p_s\}$ *such that* $p_s((\chi_i, \zeta_i)) \leftarrow \frac{C((\chi_i,\zeta_i))}{W}$;
3. *Reverse the sequence* $\{p_s\}$;
4. *For* $j = 1, \ldots, p$, *compute* $q_j \leftarrow \sum_{i=1}^{j} p_s((\chi_i, \zeta_i))$;
5. *For* $t = 1, \ldots, n_s$, *do*
   (a) *If* $t = 1$ *output* (χ_1, ζ_1), *the chromosome with least cost. End.*
   (b) *Else*
      i. *Choose a random* $r \in [0, 1]$;
      ii. *Output* (χ_k, ζ_k) *such that* $q_{k-1} < r < q_k$. *End.*

The algorithm respects the requirement that chromosomes with least cost are selected more often. For crossover we use *tournament selection*, where we input three randomly chosen chromosomes in the current population and select the two with least cost. If all three have identical cost, then select the first two chosen. Selection of chromosomes for asexual reproduction is at random from the current population.

4 Traceback

In many ways, cost functions are a large part of a GA. But the reproduction methods often specify that a random generator is chosen, so reducing the number of possible choices of generator may serve to guide the GA. We give a possible approach to reducing this number and term it *traceback*. In brief, we take the problem instance given by the pair (a, b) and use a and b to determine properties of a feasible solution $(x, y) \in \mathcal{M}((a, b))$ to the instance. This approach exploits the "geometry" of the search space by tracking the process of reduction of E to its normal form in V_n and proceeds as follows:

Recall Y and Z respectively denote the set of generators of the parabolic subgroups $G(Y)$ and $G(Z)$. Suppose we have a chromosome (χ, ζ) at some stage of the GA computation. Form the expression $E = \chi a \zeta b^{-1}$ associated to the given instance of the DCSP and label each generator from χ and ζ with its position in the product $\chi\zeta$. Then reduce E to its normal form $\overline{E}$; during reduction the labels travel with their associated generators. As a result some generators from χ or ζ may be cancelled or not, and the set of labels of the non-cancelled generators of χ and ζ give the original positions.

The generators in V_n which commute mean that the chromosome may be split into *blocks* $\{\beta_i\}$. Each block is formed from at least one consecutive generator of χ and ζ which move together under reduction of E. Let B be the set of all blocks from the above process. Now a block $\beta_m \in B$ and a position q (which we call the *recommended position*) at either the left or right end of that block are randomly chosen. Depending upon the position chosen, take the subword δ between either the current and next block β_{m+1} or the current and prior block β_{m-1} (if available). If there is just one block, then take δ to be between β_1 and the end or beginning of $\overline{E}$.

Then identify the word χ or ζ from which the position q originated and its associated generating set $S = Y$ or $S = Z$. The position q is at either the left or right end of the chosen block. So depending on the end of the block chosen, randomly select the inverse of a generator from $RF(\delta) \cap S$ or $FL(\delta) \cap S$. Call this the *recommended generator* g. Note if both χ and ζ are entirely cancelled (and so B is empty), we return a random recommended generator and position.

With these, the insertion algorithm inserts the inverse of the generator on the appropriate side of the recommended position in χ or ζ. In the cases of substitution and deletion, we substitute the recommended generator or delete the generator at the recommended position. We now give an example for the DCSP on V_{10} with the two parabolic subgroups of $V(Y) = V_7$ and $V(Z) = V_{10}$.

Example of Traceback on a Given Instance Take the short DCSP instance

$$(a, b) = (x_2^2 x_3 x_4 x_5 x_4^{-1} x_7 x_6^{-1} x_9 x_{10},\ x_2^2 x_4 x_5 x_4^{-1} x_3 x_7 x_6^{-1} x_{10} x_9)$$

and let the current chromosome be $(\chi, \zeta) = (x_3 x_2^{-1} x_3^{-1} x_5 x_7,\ x_5 x_2 x_3 x_7^{-1} x_{10})$. Represent the labels of the positions of the generators in χ and ζ by the following numbers immediately above each generator:

$$\begin{array}{ccccc|ccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ x_3 & x_2^{-1} & x_3^{-1} & x_5 & x_7 & x_5 & x_2 & x_3 & x_7^{-1} & x_{10} \end{array}$$

Forming E and reducing it to its Knuth-Bendix normal form gives

$$\overline{E} = \begin{array}{cccccccccccccc} 0 & 1 & 2 & & & & & 3 & & & & 4 & & \\ x_3 & x_2^{-1} & x_3^{-1} & x_2 & x_2 & x_3 & x_2^{-1} & x_5 & x_4 & x_5 & x_4^{-1} & x_7 & x_7 & \\ & 5 & & 8 & & & & & & & 9 & & & \\ x_6^{-1} & x_5 & x_4 & x_7^{-1} & x_6 & x_5^{-1} & x_4^{-1} & x_7^{-1} & x_9 & x_{10} & x_{10} & x_9^{-1} & x_{10}^{-1} & \end{array}$$

which contains eight remaining generators from (χ, ζ). Take cost to be $C((\chi, \zeta)) = l(\overline{E}) = 26$, the number of generators in $\overline{E}$ above. There are three blocks for χ:

$$\beta_1 = \begin{array}{ccc} 0 & 1 & 2 \\ x_3 & x_2^{-1} & x_3 \end{array},\ \beta_2 = \begin{array}{c} 3 \\ x_5 \end{array},\ \beta_3 = \begin{array}{c} 4 \\ x_7 \end{array}$$

and three for ζ:

$$\beta_4 = \begin{array}{c} 5 \\ x_5 \end{array},\ \beta_5 = \begin{array}{c} 8 \\ x_7^{-1} \end{array},\ \beta_6 = \begin{array}{c} 9 \\ x_{10} \end{array}$$

Suppose we choose position eight, which is in ζ and is block β_5. This is a block of length one; we may take the word to the left or the right as our choice for δ.

Suppose we choose the word to the right, so $\delta = x_6 x_5^{-1} x_4^{-1} x_7^{-1} x_9 x_{10}$ and in this case, $S = \{x_1, \ldots, x_{10}\}$. So we choose a random generator from $FL(\delta) \cap S = \{x_6, x_9\}$. Choose $g = x_6^{-1}$ and so ζ becomes $\zeta' = x_5 x_2 x_3 x_7^{-1} x_6^{-1} x_{10}$, with $\chi' = \chi$. The cost becomes $C((\chi', \zeta')) = l(\overline{\chi' a \zeta' b^{-1}}) = 25$. Note that we could have taken any block and the permitted directions to create δ. In this case, there are eleven choices of δ, clearly considerably fewer than the total number of subwords of $\overline{E}$. Traceback provides a significant increase in performance over merely random selection (this is easily calculated in the above example to be by a factor of 38).

5 Setup of the Genetic Algorithm

5.1 Specification of Output Alphabet

Let $n = 2m$ for some integer $m > 1$. Define the subsets of generators $Y = \{x_1, \ldots, x_{m-1}\}$, $Z = \{x_{m+2}, \ldots, x_n\}$ and two corresponding parabolic subgroups $G(Y) = \langle Y \rangle, G(Z) = \langle Z \rangle$. Clearly $G(Y)$ and $G(Z)$ commute as groups: if we take any $m > 1$ and any words $x_y \in G(Y)$, $x_z \in G(Z)$ then $x_y x_z = x_z x_y$. We direct the interested reader to [5] for information on the importance of the preceding statement. Given an instance (a, b) of the DCSP with parabolic subgroups as above, we will seek a representative for each of the two words $x \in G(Y)$ and $y \in G(Z)$ that are a solution to the DCSP. Let us label this problem (P).

5.2 The Algorithm and its Parameters

Given a chromosome (χ, ζ) we choose crossover to act on either χ or ζ at random, and fix the other component of the chromosome. Insertion is performed according to the position in χ or ζ given by traceback and substitution is with a random generator, both such that if the generator chosen cancels with a neighbouring generator from the word then another random generator is chosen. We choose to use pseudo-normal form for all chromosomes to remove all redundant generators while preserving the internal ordering of (χ, ζ).

By experiment, GA behaviour and performance is mostly controlled by the *parameter set* chosen. A parameter set is specified by the population size p and numbers of children begat by each reproduction algorithm. The collection of numbers of children is given by a multiset of non-negative integers $P = \{p_i\}$, where $\sum p_i = p$ and each p_i is given, in order, by the number of crossovers, substitutions, deletions, insertions, selections and random chromosomes. The GA is summarised by the following algorithm:

Algorithm 2 (GA for DCSP)

INPUT: *The parameter set, words a, b and their lengths $l(a), l(b)$, suicide control σ, initial length L_I*

OUTPUT: *A solution (χ, ζ) or timeout; i, the number of populations*

1. *Generate the initial population P_0, consisting of p random (unreduced) chromosomes (χ, ζ) of initial length L_I;*
2. *$i \leftarrow 0$;*
3. *Reduce every chromosome in the population to its pseudo-normal form.*
4. *While $i < \sigma$ do*
   (a) *For $j = 1, \ldots, p$ do*
      i. *Reduce each pair $(\chi_j, \zeta_j) \in P_i$ to its pseudo-normal form $(\tilde{\chi_j}, \tilde{\zeta_j})$;*
      ii. *Form the expression $E = \tilde{\chi_j}\, a\, \tilde{\zeta_j}\, b^{-1}$;*
      iii. *Perform the traceback algorithm to give $C((\chi_j, \zeta_j))$, recommended generator g and recommended position q;*

(b) *Sort current population* P_i *into least-cost-first order and label the chromosomes* $(\tilde{\chi_1}, \tilde{\zeta_1}), \ldots, (\tilde{\chi_p}, \tilde{\zeta_p})$*;*
(c) *If the cost of* $(\tilde{\chi_1}, \tilde{\zeta_1})$ *is zero then return solution* (χ_1, ζ_1) *and* END.
(d) $P_{i+1} \leftarrow \emptyset$*;*
(e) *For* $j = 1, \ldots, p$ *do*
  i. *Using the data obtained in step* $4(a)(iii)$*, perform the appropriate reproductive algorithm on* $(\tilde{\chi_j}, \tilde{\zeta_j})$ *and denote the result* (χ'_j, ζ'_j)*;*
  ii. $P_{i+1} \leftarrow P_{i+1} \cup \{(\chi'_j, \zeta'_j)\}$*;*
(f) $i \leftarrow i + 1$.

5. *Return failure.* END.

The positive integer σ is an example of a *suicide control*, where the GA stops (suicide) if more than σ populations have been generated. In all cases here, σ is chosen by experimentation; GA runs that continued beyond σ populations were unlikely to produce a successful conclusion. By deterministic search we found a population size of $p = 200$ and parameter set $P = \{5, 33, 4, 128, 30, 0\}$ for which the GA performs well when $n = 10$. We observed that the GA exhibits the well-known common characteristic of sensitivity to changes in parameter set; we consider this in future work. We found an optimal length of one for each word in our initial population, and now devote the remainder of the paper to our results of testing the GA and analysis of the data collected.

5.3 Method of Testing

We wished to test the performance of the GA on "randomly generated" instances of problem (P). Define the length of an instance of (P) to be the set of lengths $\{l(\overline{a}), l(\overline{x}), l(\overline{y})\}$ of words $a, x, y \in V_n$ used to create that instance. Each of the words a, x and y are generated by simple random walk on V_n. To generate a word $\overline{u}$ of given length $k = l(\overline{u})$ firstly generate the unreduced word u_1 with unreduced length $l(u_1) = k$. Then if $l(\overline{u_1}) < k$, generate u_2 of unreduced length $k - l(\overline{u_1})$, take $u_1 u_2$ and repeat this procedure until we produce a word $u = u_1 u_2 \ldots\ u_r$ with $l(\overline{u})$ equal to the required length k.

We identified two key input data for the GA: the length of an instance of (P) and the group rank, n. Two types of tests were performed, varying these data:

1. Test of the GA with long instances while keeping the rank small;
2. Test of the GA with instances of moderate length while increasing the rank.

The algorithms and tests were developed and conducted in GNU C++ on a Pentium IV 2.53 GHz computer with 1GB of RAM running Debian Linux 3.0.

5.4 Results

Define the *generation count* to be the number of populations (and so iterations) required to solve a given instance; see the counter i in Algorithm 2. We present the results of the tests and follow this in section 5.5 with discussion of the results.

Increasing Length We tested the GA on eight randomly generated instances (I1)–(I8) with the rank of V_n set at $n = 10$. The instances (I1)–(I8) were generated beginning with $l(\overline{a}) = 128$ and $l(\overline{x}) = l(\overline{y}) = 16$ for instance (I1) and progressing to the following instance by doubling the length $l(\overline{a})$ or both of the lengths $l(\overline{x})$ and $l(\overline{y})$. The GA was run ten times on each instance and the mean runtime $\overline{t}$ in seconds and mean generation count $\overline{g}$ across all runs of that instance was taken. For each collection of runs of an instance we took the standard deviation σ_g of the generation counts and the mean time in seconds taken to compute each population. A summary of results is given in Table 1.

Table 1. Results of increasing instance lengths for constant rank $n = 10$.

Instance	$l(\overline{a})$	$l(\overline{x})$	$l(\overline{y})$	$\overline{g}$	$\overline{t}$	σ_g	sec/gen
I1	128	16	16	183	59	68.3	0.323
I2	128	32	32	313	105	198.5	0.339
I3	256	64	64	780	380	325.5	0.515
I4	512	64	64	623	376	205.8	0.607
I5	512	128	128	731	562	84.4	0.769
I6	1024	128	128	1342	801	307.1	0.598
I7	1024	256	256	5947	5921	1525.3	1.004
I8	2048	512	512	14805	58444	3576.4	3.849

Increasing Rank These tests were designed to keep the lengths of computed words relatively small while allowing the rank n to increase. We no longer impose the condition of $l(\overline{x}) = l(\overline{y})$. Take s to be the arithmetic mean of the lengths of $\overline{x}$ and $\overline{y}$. Instances were constructed by taking $n = 10, 20$ or 40 and generating random a of maximal length 750, random x and y of maximal length 150 and then reducing the new $b = xay$ to its normal form $\overline{b}$.

We then ran the GA once on each of 505 randomly generated instances for $n = 10$, with 145 instances for $n = 20$ and 52 instances for $n = 40$. We took the time t in seconds to produce a solution and the respective generation count g. The data collected is summarised on Table 2 by grouping the length s of instance into intervals of length fifteen. For example, the range 75–90 means all instances where $s \in [75, 90)$. Across each interval we computed the means $\overline{g}$ and $\overline{t}$ along with the standard deviation σ_g. We now give a brief discussion of the results and some conjectures, and then conclude our work.

5.5 Discussion and Conclusion

Firstly, the mean times given on Tables 1 and 2 depend upon the time complexity of the underlying algebraic operations. We conjecture for $n = 10$ that these have time complexity no greater than $O(k \log k)$ where k is the mean length of all words across the entire run of the GA that we wish to reduce.

Table 1 shows we have a good method for solving large scale problems when the rank is $n = 10$. By Table 2 we observe the GA operates very well in most

Table 2. Results of increasing rank from $n = 10$ (upper rows) to $n = 20$ (centre rows) and $n = 40$ (lower rows).

s	15–30	30–45	45–60	60–75	75–90	90–105	105–120	120–135	135–150
$\overline{g}$	227	467	619	965	1120	1740	1673	2057	2412
$\overline{t}$	44	94	123	207	244	384	399	525	652
$\overline{g}$	646	2391	2593	4349	4351	8585	8178	8103	10351
$\overline{t}$	251	897	876	1943	1737	3339	3265	4104	4337
$\overline{g}$	1341	1496	2252	1721	6832	14333	14363	-	-
$\overline{t}$	949	1053	836	1142	5727	10037	11031	-	-

cases across problems where the mean length of x and y is less than 150 and rank at most forty. Fixing s in a given range, the mean generation count increases at an approximately linearithmic rate as n increases. This seems to hold for all n up to forty, so we conjecture that for a mean instance of problem (P) with given rank n and instance length s the generation count for an average run of the GA lies between $O(sn)$ and $O(sn \log n)$. This conjecture means the GA generation count depends linearly on s (for brevity, we omit the statistical evidence here).

As n increases across the full range of instances of (P), increasing numbers of suicides tend to occur as the GA encounters increasing numbers of local minima. These may be partially explained by observing traceback. For n large, we are likely to have many more blocks than for n small (as the likelihood of two arbitrary generators commuting is larger). While traceback is much more efficient than a purely random method, there are more chances to read δ between blocks. Indeed, there may be so many possible δ that it takes many GA iterations to reduce cost. By experience of this situation, non-asexual methods of reproduction bring the GA out of some local minima. Consider the following typical GA output, where the best chromosomes from populations 44 and 64 (before and after a local minimum) are:

```
Gen 44 (c = 302) : x = 9 6 5 6 7 4 5 -6 7 5 -3 -3 (l = 12)

y = -20 14 12 14 -20 -20 (l = 6)

Gen 64 (c = 300) : x = 9 8 1 7 6 5 6 7 4 5 -6 7 9 5 -3 -3 (l = 16)

y = 14 12 12 -20 14 15 -14 -14 -16 17 15 14 -20 15 -19 -20 -20 -19
-20 18 -17 -16 (l = 22)
```

In this case, cost reduction is not made by a small change in chromosome length, but by a large one. We observe that the cost reduction is made when a chromosome from lower in the ordered population is selected and then mutated, as the new chromosome at population 64 is far longer. In this case it seems traceback acts as a topological sorting method on the generators of the equation E, giving complex systems of cancellation in E which result in a cost deduction greater than one. This suggests that finetuning the parameter set to focus more

on reproduction lower in the population and reproduction which causes larger changes in word length may improve performance. Indeed, [3] conjectures that

> "It seems plausible to conjecture that sexual mating has the purpose to overcome situations where asexual evolution is stagnant."
>
> **Bremermann [3, p. 102]**

This implies the GA performs well in comparison to asexual hillclimbing methods. Indeed, this is the case in practice: by making appropriate parameter choices we may simulate such a hillclimb, which experimentally encounters many more local minima. These local minima seem to require substantial changes in the form of χ and ζ (as above); this cannot be done by mere asexual reproduction.

Meanwhile, coupled with a concept of "growing" solutions, we have at least for reasonable values of n an indication of a good underlying deterministic algorithm based on traceback. Indeed, such deterministic algorithms were developed in [2] as the result of analysis of experimental data in our work. This hints that the search space has a "good" structure and may be exploited by appropriately sensitive GAs and other artificial intelligence technologies in our framework.

References

1. R. F. Booth, D. Y. Bormotov, A. V. Borovik, Genetic Algorithms and Equations in Free Groups and Semigroups, Contemp. Math. **349** (2004), 63–80.
2. A. V. Borovik, E. S. Esyp, I. V. Kazatchkov, V. N. Remeslennikov, Divisibility Theory and Complexity of Algorithms for Free Partially Commutative Groups, Contemp. Math. **378** (Groups, Languages, Algorithms), 2005.
3. H. J. Bremermann, Optimization Through Evolution and Recombination, Self-Organizing Systems (M. C. Yovits et al., eds.), Washington, Spartan Books (1962), 93–106.
4. J. Holland, Adaptation in Natural and Artificial Systems (5th printing), MIT Press, Cambridge, Massachusetts, 1998.
5. K. -H. Ko, Braid Group and Cryptography, 19th SECANTS, Oxford, 2002.
6. D. Knuth, P. Bendix, Simple Word Problems in Universal Algebra, Computational Problems in Abstract Algebras (J. Leech, ed.), Pergamon Press 1970, 263–297.
7. A. D. Miasnikov, Genetic Algorithms and the Andrews-Curtis Conjecture, Internat. J. Algebra Comput. **9** (1999), no. 6, 671–686.
8. A. D. Miasnikov, A. G. Myasnikov, Whitehead Method and Genetic Algorithms, Contemp. Math. **349** (2004), 89–114.
9. Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs (3rd rev. and extended ed.), Springer-Verlag, Berlin, 1996.
10. L. VanWyk, Graph Groups are Biautomatic, J. Pure Appl. Algebra **94** (1994), no. 3, 341–352.
11. A. Vershik, S. Nechaev, R. Bikbov, Statistical Properties of Braid Groups in Locally Free Approximation, Comm. Math. Phys. **212** (2000), 59–128.
12. C. Wrathall, The Word Problem for Free Partially Commutative Groups, J. Symbolic Comp. **6** (1988), 99–104.
13. C. Wrathall, Free partially commutative groups, Combinatorics, Computing and Complexity (Tianjing and Beijing, 1988) 195–216, Math. Appl (Chin. Ser. 1) Kluwer Acad. Publ., Dordrecht, 1989.